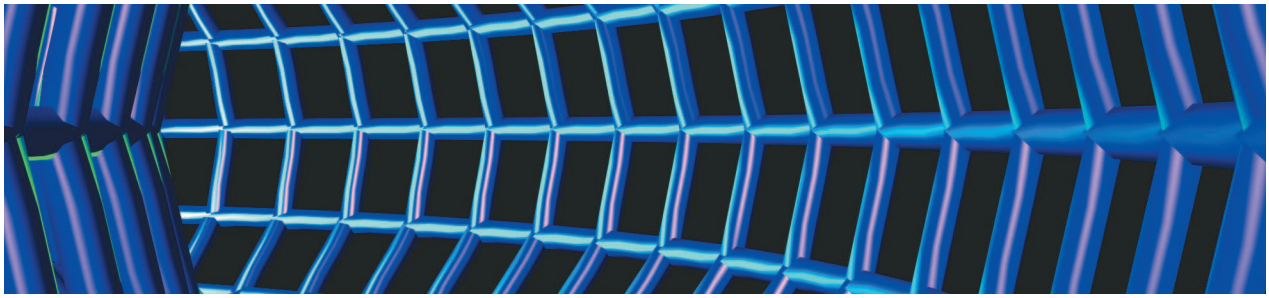
10-2012

Defeating SQL injection

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Hee Beng Kuan TAN



Defeating SQL Injection

Lwin Khin Shar and Hee Beng Kuan Tan, *Nanyang Technological University, Singapore*

The best strategy for combating SQL injection, which has emerged as the most widespread website security risk, calls for integrating defensive coding practices with both vulnerability detection and runtime attack prevention methods.

Structured Query Language injection is a code injection technique commonly used to attack websites in which the attacker inserts SQL characters or keywords into a SQL statement via unrestricted user input parameters to change the intended query's logic.[1] This threat exists in any Web application that accesses a database via SQL statements constructed with external input data. By manipulating this data to modify the statements, an attacker can cause the application to issue arbitrary SQL commands and thereby compromise the database.

The Open Web Application Security Project (OWASP) ranks SQL injection as the most widespread website security risk (www.owasp.org/index.php/Top_10). In 2011, the National Institute of Standards and Technology's National Vulnerability Database (nvd.nist.gov) reported 289 SQL injection vulnerabilities (7 percent of all vulnerabilities) in websites, including those of IBM, Hewlett-Packard, Cisco, WordPress, and Joomla. In December 2011, SANS Institute security experts reported a major SQL injection attack (SQLIA) that affected approximately 160,000 websites using Microsoft's Internet Information Services (IIS), ASP.NET, and SQL Server frameworks (isc.sans.org/diary/SQL+Injection+Attack+happening+ATM/12127).

Inadequate validation and sanitization of user inputs make websites vulnerable to SQL injection, and researchers have proposed various ways to address this problem, ranging from simple static analysis to complex dynamic analysis. In 2006, William Halfond, Jeremy Viegas, and Alessandro Orso[2] evaluated then-available techniques and called for more precise solutions. In reviewing work during the past decade, we found that developers can effectively combat SQL injection using the right combination of state-of-the art methods. However, they must develop a better understanding of SQL injection and how to practically integrate current defenses.

INSECURE CODING PRACTICES

SQL is the standard language for accessing database servers, including MySQL, Oracle, and SQL Server.[1] Web programming languages such as Java, ASP.NET, and PHP provide various methods for constructing and executing SQL statements, but, due to a lack of training and development experience, application developers often misuse these methods, resulting in SQL injection vulnerabilities (SQLIVs).

Developers commonly rely on dynamic query building with string concatenation to construct SQL statements. During runtime, the system forms queries with inputs directly received from external sources. This method makes it possible to build different queries based on varying conditions set by users. However, as this is the cause of many SQLIVs, some developers opt to use parameterized queries or stored procedures. While these methods are more secure, their inappropriate use can still result in vulnerable code. In the PHP code examples below, `name` and `pwd` are the "varchar" type columns and `id` is the "integer" type column of a `user` database table.

Absence of checks. The most common and serious mistake developers make is using inputs in SQL statements without any checks. The following PHP code is an example of such a dynamic SQL statement:

0018-9162/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

```
$query = "SELECT info FROM user WHERE name =
  '$_GET["name"]' AND pwd = '$_GET["pwd"]'";
```

Attackers can use *tautologies* to exploit this insecure practice. In this case, by supplying the value `x' OR '1'='1` to the input parameter `name`, an attacker could access user information without a valid account because the WHERE-clause condition becomes

```
WHERE name = 'x' OR '1'='1' AND …";
```

which the system will evaluate to be true.

Insufficient escaping. If a developer escapes special characters meaningful to a SQL parser, the parser will not interpret them as SQL commands. For example, the above tautology-based attack could be prevented by escaping the ' character (to avoid its being interpreted as a string delimiter) from the inputs. However, many developers are either not aware of the full list of characters that have special meanings to the SQL parser or they are not familiar with the proper usage patterns.

Consider the following PHP code, `mysql_real_escape_string`, which is a function used to escape MySQL special characters:

```
$name = mysql_real_escape_string($_GET["name"]);
$query = "SELECT info FROM user WHERE pwd LIKE
  '%$pwd%'";
```

The function `mysql_real_escape_string` would protect SQL statements that do not use pattern-matching database operators such as `LIKE`, `GRANT`, and `REVOKE`. In this case, however, an attacker could include the additional wildcard characters `%` and `_` in the password field to match more password characters than the beginning and end characters because `mysql_real_escape_string` does not escape wildcard characters.

Absence of data type checks. Another error that developers make is failing to check data types before constructing SQL statements. Instead, they often apply programming language or database-provided sanitization functions such as `addslashes` and `mysql_real_escape_string` to the input parameters before using them in SQL statements.

However, when the query is to access the database columns of numeric data and other non-text-based data types, a SQLIA need not contain the escaped/sanitized characters. For example, the following PHP code shows a SQL statement for which a tautology-based attack could be conducted by supplying the value `1 OR 1=1` to the parameter `id`:

```
$id = mysql_real_escape_string($_GET["id"]);
$query = "SELECT info FROM user WHERE id =
  $id";
```

For such queries, instead of escaping characters, developers should use a data type check—for example, `if(is_numeric($id))`—to prevent SQLIAs.

Absence or misuse of delimiters in query strings. When constructing a query string with inputs, a programmer must use proper delimiters to indicate the input's data type. The absence or misuse of delimiters could enable SQL injection even in the presence of thorough input validation, escaping, and type checking. For example, the following PHP code does not include delimiters to indicate the input string used in the SQL statement:

```
$name = mysql_real_escape_string($_GET["name"]);
$query = "SELECT info FROM user WHERE name =
  $name";
```

In this case, when the database server has the automatic type conversion function enabled, an attacker could use an alternate encoding method that circumvents input sanitization routines. For instance, if the attacker supplies the encoded HEX string `0x270x780x270x200x4f0x520x200x310x3d0x31` to the parameter `name`, the database parser may convert it to the "varchar" value, resulting in the tautology string `'x' OR 1=1`. Because the conversion occurs in the database, the server program's escaping function would not detect any special characters encoded in the HEX string.

Improper parameterized queries or stored procedures. Most developers believe that SQL injection is impossible when using parameterized queries or stored procedures to run SQL statements. Although this is generally true, some developers are not aware that SQL injection is still possible if parameterized query strings or stored procedures accept nonparameterized inputs.

Consider, for example, the following PHP code:

```
$query = "SELECT info FROM user WHERE name =
  ?"."ORDER BY '$_GET["order"]'";
$stmt = $dbo->prepare($query);
$stmt->bindParam(1, $_GET["name"]);
$stmt->execute();
```

Although an attacker could not conduct a SQLIA through the parameter `name`, SQL injection is still possible through `order`, which is not parameterized. An attacker could inject piggy-backed query attacks—malicious queries attached to the original query—such as `ASC; DROP TABLE user; --` into the parameter order.

SQL INJECTION DEFENSES

SQL injection defense methods can be broadly classified into three types: defensive coding, SQLIV detection, and SQLIA runtime prevention. Table 1 compares the strengths and weaknesses of various approaches in each category.

Table 1. Comparison of SQL injection defenses.

Defense type	Defense	User involvement	Vulnerability locating	Verification assistance	Code modification	Generate test suite	Usage stage	Infrastructure
Defensive coding	Manual defensive coding practices	Very high	No	No	Manual	No	Development	Developer training
	SQL DOM	High	No	No	Manual	No	Development	Developer training
	Parameterized query insertion	Medium	No	No	Automated	No	Testing and debugging	Tool for code replacement
SQLIV detection	SQLUnitGen	Medium	Automated	Unit test reports	No	Yes	Testing and debugging	Static analysis tool
	MUSIC	Very high	Manual inspection	Test inputs that expose the weaknesses of implemented defense mechanisms	Manual	Yes	Testing and debugging	Manual tests
	Vulnerability and attack injection	Low	Manual inspection	Test inputs that expose the weaknesses of implemented defense mechanisms	Automated	Yes	Testing and debugging	Injection tool
	SUSHI	Low	Automated	Path conditions that lead to SQLIVs	Automated	Yes	Testing and debugging	Symbolic execution engine
	Ardilla	Low	Automated	Concrete attacks	Automated	Yes	Testing and debugging	Concolic execution engine
	String analyzer	Medium	Automated	Static dataflow traces	No	No	Code verification	Static string analysis tool
	PhpMinerI	Low	Automated	Statistics of sanitization methods implemented	No	No	Code verification	Static analysis and data mining tool
Runtime SQLIA prevention	SQLrand	High	No	No	Manual	No	Deployment	Runtime checker
	AMNESIA	Low	No	Static dataflow traces	Automated	No	Deployment	Static analysis tool and runtime checker
	SQLCheck	Low	No	No	No	No	Deployment	Runtime checker
	WASP	Low	No	No	Automated	No	Deployment	Instrumentation tool and runtime checker
	SQLProb	High	No	No	No	No	Deployment	Runtime checker
	CANDID	Low	No	No	Automated	No	Deployment	Instrumentation tool and runtime checker

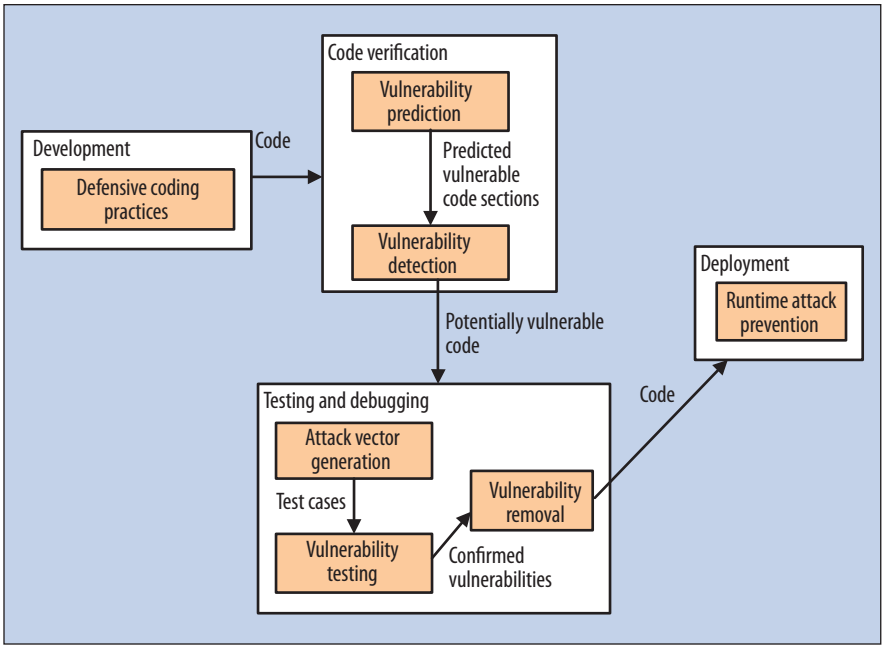


Figure 1. Web application developers could overcome the shortcomings of individual SQL injection methods by combining various schemes.

Developers could overcome the shortcomings of individual methods by combining schemes, as Figure 1 shows.

Defensive coding

Defensive coding is a straightforward solution, as SQLIVs are the direct consequence of developers' insecure coding practices.

Manual defensive coding practices. Many security reports, such as OWASP's SQL Injection Prevention Cheat Sheet (http://owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet) and Chris Anley's white paper,[1] provide useful manual defensive coding guidelines.

Parameterized queries or stored procedures. Replacing dynamic queries with properly coded parameterized queries or stored procedures would force developers to first define the SQL code's structure before including parameters to the query. Because parameters are bound to the defined SQL structure, it is not possible to inject additional SQL code.

Escaping. If dynamic queries cannot be avoided, escaping all user-supplied parameters is the best option. However, as insufficient or improper escaping practices are common, developers should identify all input sources to realize the parameters that need escaping, follow database-specific escaping procedures, and use standard escaping libraries instead of custom escaping methods.

Data type validation. In addition to escaping, developers should use data type validation. Validating whether an input is string or numeric could easily reject type-mismatched inputs. This could also simplify the escaping process because validated numeric inputs need no further cleansing action and could be safely used in queries.

White list filtering. Developers often use *black list filtering* to reject known bad special characters such as ' and ; from the parameters to avoid SQL injection. However, accepting only inputs known to be legitimate is safer. This filtering approach is suitable for well-structured data such as email addresses, dates, zip codes, and Social Security numbers. Developers could keep a list of legitimate data patterns and accept only matching input data.

SQL DOM. Although manual defensive coding practices are the best way to defeat SQL injection, their application is labor-intensive and error-prone. To alleviate these problems, Russell McClure and Ingolf Krüger[3] created SQL DOM, a set of classes that enables automated data type validation and escaping. Developers provide their own database schema and construct SQL statements using its APIs. SQL DOM is especially useful when developers need to use dynamic queries instead of parameterized queries for greater flexibility. However, they can only use it with new software projects, and they must learn a new query-development process.

Parameterized query insertion. An automated vulnerability removal approach finds potentially vulnerable (dynamic) SQL statements in programs and replaces them with parameterized SQL statements.[4] For example, this approach would replace the PHP code

```
$rs = mysql_query("SELECT info FROM user WHERE
  id = '$id');
```

with the following code:

```
$dbh=new PDO("mysql:host=xxx;dbname=xxx;","root",
  "pwd");
$PSinput00[] = Array();
$PSquery00 = "SELECT info FROM user WHERE id =
  ?";
$PSInput00[] = $id;
$stmt = $dbh->prepare($query);
$i = 1;
foreach($PSinput00 as $input){
   $stmt->bindParam($i++, $input);
}
$rs = $stmt->execute();
```

A shortcoming of this method is that it only works on SQL structures built with explicit strings; developers must incorporate program analysis techniques to deduce SQL structures built with data objects or through function calls.

SQLIV detection

Researchers have developed several methods to detect SQLIVs.

Code-based vulnerability testing. This approach generally aims to generate an adequate test suite for detecting SQLIVs. However, it does not explicitly find vulnerable program points, necessitating manual inspection.

SQLUnitGen[5] is a prototype tool that uses static analysis to track user inputs to database access points and generate unit test reports containing SQLIA patterns for these points.

MUSIC (mutation-based SQL injection vulnerability checking)[6] uses nine mutation operators to replace original queries in a Web program with mutated queries. José Fonseca, Marco Vieira, and Henrique Madeira[7] developed a tool that automatically injects SQLIVs into Web programs and generates SQLIAs. Both tools assess the effectiveness of the security mechanisms implemented in the application under test based on the injected mutants/vulnerabilities detected.

Concrete attack generation. This type of approach uses state-of-the-art symbolic execution techniques to automatically generate test inputs that actually expose SQLIVs in a Web program.

Symbolic execution generates test inputs by solving the constraints imposed on the inputs along the path to be exercised. Traditionally, symbolic-execution-based approaches use constraint solvers that only handle numeric operations. Because inputs to Web applications are by default strings, if a constraint solver can solve myriad string operations applied to inputs, developers could use symbolic execution to both detect the vulnerability of SQL statements that use inputs and generate concrete inputs that attack them.

Xiang Fu and Chung-Chih Li developed a vulnerability detection tool consisting of JavaSye, a symbolic execution engine, and SUSHI, a powerful hybrid (numeric and string) constraint solver.[8] SUSHI solves path conditions that lead to SQL statements and extracts test inputs containing SQLIAs from the solution pool. As the "SUSHI Constraint Solver" sidebar describes, if the tool generates such a test input, the corresponding SQL statement is vulnerable.

Although effective, symbolic execution alone is generally not scalable to large programs due to path explosion. Researchers have thus proposed various solutions to improve code coverage. Ardilla[9] incorporates concrete execution into symbolic execution, using randomized concrete test inputs to exercise program paths that constraint solvers cannot symbolically solve. SWAT[10] uses a search-based algorithm that formulates test input adequacy criteria as fitness functions. It uses these functions to compare pooled

SUSHI Constraint Solver

Consider the following snippet of vulnerable PHP code:

```
1 $id = addslashes($_COOKIE["id"]);
2 $query = "SELECT info FROM user WHERE ";
3 if($id!=null) {
4    $query .= "id = $id";    //path 1
  } else {
5    $name = addslashes($_GET["name"]);
6    $order = addslashes($_GET["order"]);
7    $query .= "name = '$name' ORDER BY
         '$order'";      //path 2
  }
8 $rs = mysql_query($query);
```

The function `addslashes()` escapes string delimiters. Therefore, a SQL injection attack is not possible through path 2 because a single-quote string delimiter is required to cancel out the delimiters (`$name`) used in the query; however, a SQLIA is possible through path 1 because the cookie parameter `id` is an integer type and thus a string delimiter is not required to create an attack.

$X_{c\to\backslash c}$ is the symbolic expression of the `addslashes()` function—which escapes the four SQL special characters ', ", \, and \0—performed on `id`. When the symbolic execution engine reaches the query execution statement at line 8, SUSHI constructs and solves the following constraint, a conjunction of two string equations in which + represents string concatenation and ≡ separates the left- and right-hand sides of the equations:

$$!(X_{c\to\backslash c} \equiv \text{null}) \wedge (\text{"id = "} + X_{c\to\backslash c} \equiv \text{id = [0-9]* OR 1=1 --})$$

The second equation asks: Is it possible to obtain a solution for X such that the string on the left-hand side is matched by the regular expression on the right-hand side? If yes, the query structure constructed with the string on the left-hand side is vulnerable because the regular expression on the right-hand side is a representation of a tautology attack (SUSHI maintains a set of regular expressions that represent different types of attack patterns). In this case, SUSHI clearly has a solution for this constraint, thereby detecting an SQL injection vulnerability:

```
1 → $id: X_{c→\c}
      Path Condition: true
2 → $id: X_{c→\c}
      $query: "SELECT …"
      Path Condition: true
3 → $id: X_{c→\c}
      $query: "SELECT …"
      Path Condition: X_{c→\c}=null
4 → $id: X_{c→\c}
      $query: "SELECT …"."id = X_{c→\c}"
      Path Condition: X_{c→\c}!=null
8 → Database access point found,
      constraint solver is invoked!
```

test inputs and then applies the best (fittest) test inputs to further explore program paths.

To the best of our knowledge, researchers have not developed search-based or AI algorithms to detect SQL injection vulnerabilities. However, it is possible to incorporate these recent techniques into symbolic-execution-based SQLIV detection—for example, eliminating false negatives resulting from code uncovered by symbolic execution.

Taint-based vulnerability detection. Researchers have formulated SQL injection as an information flow integrity problem.[11] As such, it can be avoided by using static and dynamic techniques to prevent tainted data (user inputs) from affecting untainted data, such as programmer-defined SQL query structures.

Several researchers have applied prominent static analysis techniques, such as flow-sensitive analysis, context-sensitive analysis, alias analysis, and interprocedural dependency analysis, to identify input sources and data sinks (database access points) and check whether every flow from a source to a sink is subject to an input validation and/or input sanitization routine.[12,13] However, these approaches suffer from one or more of the following limitations: they do not precisely model the semantics of such routines, do not consider input validation using predicates, fail to specify vulnerability patterns, or require user intervention to state the taintedness of external or library functions that inputs pass through. All these limitations could result in false negatives or positives.

Gary Wassermann and Zhendong Su[14] used context-free grammars to model the effects of input validation and sanitization routines. Their technique checks whether SQL queries syntactically confine the string values returned from those routines and, if so, automatically concludes that the routines used are correctly implemented (and vice versa). Wassermann and Su's approach would not miss any vulnerability, but it does not precisely handle some of the complex string operations, and its conservative assumptions might result in false positives.

Data-mining-based vulnerability prediction. PhpMinerI[15] mines static code attributes that represent the characteristics of input sanitization routines implemented in Web programs. It then feeds the mined attributes and the associated vulnerability information of existing programs to lightweight classifications for building vulnerability predictors.

Consider, for example, the following PHP code snippet:

```
$id = addslashes($_COOKIE["id"]);
$query = "SELECT info FROM user WHERE ";
if($id!=null) {
  $query .= "id = $id";      //path 1
} else {
  $name = addslashes($_GET["name"]);
  $order = addslashes($_GET["order"]);
  $query .= "name = '$name' ORDER BY
        '$order'";      //path 2
}
$rs = mysql_query($query);
```

Running PhpMinerI on this code would produce the following vulnerability predictor in the form of a classification tree:

```
sql_sanit < 0 : Vulnerable
sql_sanit  ≥ 0
|      dbattr_num < 0 : Not-Vulnerable
|      dbattr_num  ≥ 0
|      | num_check < 0 : Vulnerable
|      | num_check  ≥ 0 : Not-Vulnerable
```

The tree indicates that a database access point is vulnerable if no database-specific sanitization routine is implemented or if there is an access to a database table's numeric column without any numeric type check in the program.

Such a probabilistic-based approach does not provide precise analysis of vulnerabilities, but it is still useful given that collecting static code attributes is easy and powerful data mining tools such as Weka (cs.waikato.ac.nz/ml/weka) are readily available. Developers could save much effort by focusing on those code sections predicted to be vulnerable, while incorporating techniques that mine control-flow and data-dependency graphs would better discriminate vulnerability signatures and improve precision in vulnerability localization.

Runtime SQLIA prevention

Researchers have developed tools and techniques that could prevent all SQLIAs by checking actual runtime against legitimate queries. However, runtime checks incur a performance penalty, and some of these approaches require code instrumentation to enable runtime checking, which might make debugging security vulnerabilities even more complex.

Randomization. SQLrand is a proposed mechanism that forces developers to construct queries using randomized SQL keywords instead of normal keywords.[16] A proxy filter intercepts queries sent to the database and de-randomizes the keywords. An attacker could not inject SQL code without the secret key to randomization.

Learning-based prevention. This type of approach uses a runtime monitoring system deployed between the application server and database server. It intercepts all queries and checks SQL keywords to determine whether the queries' syntactic structures are legitimate (programmer-intended) before the application sends them to the database.

User specification. Specification-based methods require developers to specify legitimate query structures using

formal language expressions such as Extended Backus-Naur Form.[17,18]

Static analysis. AMNESIA (Analysis for Monitoring and NEutralizing SQL Injection Attacks)[19] uses static analysis to deduce valid queries that might appear at each database access point in Web programs via isolation of tainted and untainted data. Another runtime SQLIA prevention technique uses a query learning approach similar to AMNESIA, but, instead of targeting query statements in a server program, it targets stored procedures in a database.[20]

Dynamic analysis. Statically inferred legitimate query structures might not be accurate, and attackers could exploit this weakness to conduct SQLIAs.[21] Researchers have thus proposed dynamic-analysis-based approaches to provide more accuracy.

SQLCheck[22] tracks tainted data at runtime by marking it with metacharacters. When a Web application invokes a query, SQLCheck learns the query's legitimate structure by excluding marked data from it. Conversely, WASP (Web application SQL-injection preventer)[23] tracks untainted data because identifying all input sources is often difficult, thus some tainted data might go undetected. Metacharacter marking requires low user effort, but it changes the data's original structure and thus might cause unpredictable errors on benign inputs.

SQLProb[24] executes a program of interest with various valid inputs to collect all possible queries that might legitimately appear during runtime. During runtime, it uses a global pairwise alignment algorithm to compare issued user queries against those in the legitimate query repository and extracts the user inputs. It then uses a SQL parser to check whether each extracted input is indeed part of the issued query's syntactic structure. SQLProb sends the query to the database only if the user input is syntactically confined. This approach requires using test inputs and assumes that the test inputs are sufficient to exercise all possible queries in the program.

CANDID[21] dynamically mines a program's legitimate query structure at each path by executing the program with valid and nonattacking inputs and, thereafter, comparing the actual issued query with the legitimate query structure mined for the same path.

To illustrate, consider again the PHP code snippet. If the runtime user input is `id←"1 OR 1=1 --"`, the input exercises path 1 and generates a query whose structure is `SELECT ? FROM ? WHERE ?=? OR ?=? --` while its corresponding candidate input (a valid input that exercises the same path as the runtime input), `id←"1"`, generates a different query structure: `SELECT ? FROM ? WHERE ?=?`. CANDID detects a SQLIA and prevents execution of the query. If the runtime user input is `id←null; name="x' OR '1'='1"; order←"ASC"`, the input exercises path 2 and generates a query whose structure is `SELECT ? FROM ? WHERE ?=? ORDER BY ?`. In this case, due to the use of escaping in source code, the attempted SQLIA does not generate a query structure different from that generated by its corresponding candidate input, `id←null; name="x"; order←"x"`. Therefore, CANDID does not consider it as an attack.

TOOL SUPPORT

To aid developers and security testers, some researchers have made their work or implementations available online.

In addition to the SQL Injection Prevention Cheat Sheet, OWASP provides the Enterprise Security API (ESAPI), a library of various security APIs for retrofitting SQL injection defense mechanisms into existing Web applications (http://owasp.org/index.php/Category:OWASP_Enterprise_Security_API), to assist in defensive coding.

For SQLIV detection, a symbolic-execution-based tool is available for download (http://people.hofstra.edu/Xiang_Fu/XiangFu/projects.php). As SUSHI is an independent solver, developers can use it for different programming languages including Java and PHP. To improve SQLIV coverage, developers can also use other symbolic execution engines in place of JavaSye, such as JavaPathFinder (http://babelfish.arc.nasa.gov/trac/jpf). The static string analysis tools for PHP (http://score.is.tsukuba.ac.jp/~minamide/phpsa) and PhpMinerI (http://sharlwinkhin.com/phpminer.html) are also available online.

Numerous off-the-shelf offerings are useful for quickly detecting the presence of SQLIVs in websites.

Downloadable runtime SQLIA prevention implementations include the static-analysis-based AMNESIA (http://www-bcf.usc.edu/~halfond/amnesia.html), which works on Java. The dynamic-analysis-based WASP is being commercialized.

Numerous off-the-shelf offerings are useful for quickly detecting the presence of SQLIVs in websites. SecuBat (secubat.codeplex.com), an open source black-box vulnerability scanner, uses a Web spider to identify test targets—for example, webpages that accept user inputs. It then launches predefined attacks against these targets and determines whether an attack was successful by evaluating the server response against attack-specific response criteria, such as SQL exceptions raised and program crashes. Other open source scanners, such as Nikto2 (http://cirt.net/nikto2) and sqlmap (http://sqlmap.org), are similar to SecuBat, but they generally require known vulnerability patterns or user intervention to conclude successful attacks.

Marco Vieira, Nuno Antunes, and Henrique Madeira[25] tested and reported on the performance of three popu-

lar commercial vulnerability scanners: HP WebInspect (www.hpenterprisesecurity.com/products/hp-fortify-software-security-center/hp-webinspect), IBM Rational (now Security) AppScan (http://www-01.ibm.com/software/awdtools/appscan), and the Acunetix Web Vulnerability Scanner (www.acunetix.com/vulnerability-scanner).

Each of the three main avenues to defeat SQL injection has its own strengths and weaknesses. Defensive coding practices will ensure secure code but are time-consuming and labor-intensive. Vulnerability detection approaches can identify most if not all SQLIVs, but they will also generate many false alarms. Runtime prevention methods can prevent SQLIAs, but they require dynamic monitoring systems. The most effective strategy calls for combining all three approaches. However, this presents two major challenges.

First, Web application developers need more extensive training to raise their awareness about SQL injection and to become familiar with state-of-the-art defenses. At the same time, they need sufficient time and resources to implement security measures. Too often, project managers pay less attention to security than to functional requirements.

Second, researchers should implement their proposed approaches and make such implementations, along with comprehensive user manuals, available either commercially or as open source. Too many existing techniques are either not publicly available or are difficult to adopt. Readily available tools would motivate more developers to combat SQL injection. In addition, researchers should find simple ways to effectively combine existing defensive schemes to overcome the limitations of individual methods rather than focusing exclusively on novel ones.

Traditionally, SQL injection was limited to personal computing environments. However, the increasing use of smartphones, tablets, and other portable devices has extended this problem to mobile and cloud computing environments, where vulnerabilities could spread much faster and become much easier to exploit. Security researchers therefore need to address additional SQLIV-related issues arising from the greater flexibility and mobility of emerging computing platforms as well as newer programming languages such as HTML5.

References

1. C. Anley, "Advanced SQL Injection in SQL Server Applications," white paper, Next Generation Security Software Ltd., 2002; www.thomascookegypt.com/holidays/pdfpkgs/931.pdf.
2. W.G.J. Halfond, J. Viegas, and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," *Proc. Int'l Symp. Secure Software Eng.* (ISSSE 06), IEEE CS, 2006; www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf.
3. R.A. McClure and I.H. Krüger, "SQL DOM: Compile Time Checking of Dynamic SQL Statements," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), ACM, 2005, pp. 88-96.
4. S. Thomas, L. Williams, and T. Xie, "On Automated Prepared Statement Generation to Remove SQL Injection Vulnerabilities," *Information and Software Technology*, Mar. 2009, pp. 589-598.
5. Y. Shin, L. Williams, and T. Xie, *SQLUnitGen: Test Case Generation for SQL Injection Detection*, tech. report TR 2006-21, Computer Science Dept., North Carolina State Univ., 2006.
6. H. Shahriar and M. Zulkernine, "MUSIC: Mutation-Based SQL Injection Vulnerability Checking," *Proc. 8th Int'l Conf. Quality Software* (QSIC 08), IEEE CS, 2008, pp. 77-86.
7. J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & Attack Injection for Web Applications," *Proc. 39th Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 09), IEEE, 2009, pp. 93-102.
8. X. Fu and C.-C. Li, "A String Constraint Solver for Detecting Web Application Vulnerability," *Proc. 22nd Int'l Conf. Software Eng. and Knowledge Eng.* (SEKE 10), Knowledge Systems Institute Graduate School, 2010, pp. 535-542.
9. A. Kieżun et al., "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), IEEE CS, 2009, pp. 199-209.
10. N. Alshahwan and M. Harman, "Automated Web Application Testing Using Search Based Software Engineering," *Proc. 26th IEEE/ACM Int'l Conference Automated Software Eng.* (ASE 11), IEEE, 2011, pp. 3-12.
11. K.J. Biba, *Integrity Considerations for Secure Computing Systems*, tech. report ESD-TR-76-372, Electronic Systems Division, US Air Force, 1977.
12. V.B. Livshits and M.S. Lam, "Finding Security Vulnerabilities in Java Programs with Static Analysis," *Proc. 14th Conf. Usenix Security Symp.* (Usenix-SS 05), Usenix, 2005; http://suif.stanford.edu/papers/usenixsec05.pdf.
13. Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," *Proc. 15th Conf. Usenix Security Symp.* (Usenix-SS 06), Usenix, 2006; http://theory.stanford.edu/~aiken/publications/papers/usenix06.pdf.
14. G. Wassermann and Z. Su, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 07), ACM, 2007, pp. 32-41.
15. L.K. Shar and H.B.K. Tan, "Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), IEEE, 2012, pp. 1293-1296.
16. S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," *Proc. 2nd Conf. Applied Cryptography and Network Security* (ACNS 04), LNCS 3089, Springer, 2004, pp. 292-302.
17. K. Kemalis and T. Tzouramanis, "SQL-IDS: A Specification-Based Approach for SQL-Injection Detection," *Proc. ACM Symp. Applied Computing* (SAC 08), ACM, 2008, pp. 2153-2158.
18. Y.-W. Huang et al., "Securing Web Application Code by Static Analysis and Runtime Protection," *Proc. 13th*

Int'l Conf. World Wide Web (WWW 04), ACM, 2004, pp. 40-52.

19. W.G.J. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks," *Proc. 3rd Int'l Workshop Dynamic Analysis* (WODA 05), ACM, 2005; www.cc.gatech.edu/~orso/papers/halfond.orso.WODA05.pdf.
20. K. We, M. Muthuprasanna, and S. Kothari, "Preventing SQL Injection Attacks in Stored Procedures," *Proc. Australian Software Eng. Conf.* (ASWEC 06), IEEE CS, 2006, pp. 191-198.
21. P. Bisht, P. Madhusudan, and V.N. Venkatakrishnan, "CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks," *ACM Trans. Information and System Security*, Feb. 2010; www.cs.illinois.edu/~madhu/tissec09.pdf.
22. Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," *Proc. 33rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 06), ACM, 2006, pp. 372-382.
23. W. Halfond, A. Orso, and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," *IEEE Trans. Software Eng.*, Jan. 2008, pp. 65-81.
24. A. Liu et al., "SQLProb: A Proxy-Based Architecture towards Preventing SQL Injection Attacks," *Proc. 24th ACM Symp. Applied Computing* (SAC 09), ACM, 2009, pp. 2054-2061.
25. M. Vieira, N. Antunes, and H. Madeira, "Using Web Security Scanners to Detect Vulnerabilities in Web Services," *Proc. 39th Ann. IEEE/IFIP Int'l. Conf. Dependable Systems and Networks* (DSN 09), IEEE, 2009, pp. 566-571.

Lwin Khin Shar *is a research student in the school of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. His research interests include software security and Web security. Shar received a BE in electrical and electronic engineering from Nanyang Technological University. He is a member of IEEE. Contact him at shar0035@ntu.edu.sg.*

Hee Beng Kuan Tan *is an associate professor of information engineering in the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. His research focuses on software security, analysis, and testing. Tan received a PhD in computer science from the National University of Singapore. He is a senior member of IEEE and a member of ACM. Contact him at ibktan@ntu.edu.sg.*